4341

Department of Electrical Engineering
Eindhoven University of Technology
Eindhoven, The Netherlands
Group Measurement and Control

PASCAL FOR PARALLEL PROCESSES, AN IMPLEMENTATION ON A MULTIPROCESSORSYSTEM

by: F.J.H.M. Meulenbroeks

Submitted in partial fulfillment of the requirements for the degree of ir. (M.Sc.). The work was carried out under directorship of prof.ir. F.J. Kylstra. Advisor: ir. C.A.M. van den Brekel.

The Department of Electrical Engineering
of the Eindhoven University of Technology
does not accept responsibility for the
content of this report.

Summary.

This report describes the design and implementation of a Pascal system as a programming language for parallel processes on a multiprocessor configuration. This configuration consists of two LSI 11/23 processors, which operate on a common memory.
The Pascal system developed is rather flexible, modular and easy to maintain. It is however rather dependent on the Pascal compiler used. Restrictions and suggestions for extensions of the system are also mentioned.

Contents.

Summary. 2
Contents. 3
Preface. 4
Introduction. 5
1. Description of the existing hard- and software. 6
1.1. Description of the hardware configuration. 6
1.2. Description of the existing software. 6
2. Problem analysis and structure of the system. 9
2.1. Design considerations. 9
2.2. General description of the structure of the system. 9
2.3. Alternatives to implement the processes. 10
2.4. Addressing in Pascal. 11
2.5. Choice of an alternative. 13
3. Implementation. 14
3.1. Implementation of the local stacks. 14
3.2. Implementation of the process descriptor. 14
3.3. Implementation of the queues. 15
3.4. Addressing the I/O page. 15
3.5. Addressing the local memory. 15
4. Error handling. 17
4.1. Detecting an error. 17
4.2. Generating an error message. 17
5. Limitations and suggestions for extensions of the system. 19
Conclusions. 20
References. 21
Appendix A: Description of the function of the preprocessor.
Appendix B: Pascal-plus user manual.
Appendix C: Generating a device common.
Appendix D: Structure of the local memory.
Appendix E: Software description.
Appendix F: Description of the compiler modification.
Appendix G: Software changes needed for hardware changes.
Appendix H: Some remarks on Pascal.
Appendix I: Modifications made in the "practicumsysteem".

Preface.

This report describes a system which can execute parallel processes, written in Pascal, on a multiprocessor system.
First (chapter 1) a description of the existing hard- and software is given. In the second chapter the general structure of the system is described. Chapter 3 deals with the implementation of the system. Emphasis is laid on the special problems caused by the hardware configuration used. Chapter 4 explains the error handling of the system, whilst in chapter 5 the limitations of the system are mentioned, and suggestions for extensions are made.
Some communication and synchronisation routines are added to Pascal. The so-created language is called Pascal-plus (PAP), and is described in appendix B. Appendix A gives a description of the function of the preprocessor, which transforms a program written in Pascal-plus, into Pascal. Appendices C and F give implementation details of some parts of the system, whilst appendix E gives a description of the software. Appendix D describes the structure of the local memory. Some software modifications needed in case of hardware changes are described in appendix G.
During the development of the system, some bugs in the Pascal compiler were detected. These are reported in appendix H. A program already written earlier, was used as test and demonstration program. Some modifications were made in this program. These are described in appendix I.

Introduction.

Within the research project ER 17, a multiprocessor system has been realized, which consists of two LSI 11/23 processors, which operate on a common memory.
Objective of the thesis project is to realize the possibility to execute simultaneous processes, written in Pascal, on this hardware configuration. For the interaction between these processes one wants to have communication and synchronisation primitives at one's disposal. Starting the processes must also be implemented.
It is assumed that the reader is familiar with the language Pascal. The formal description of this language can be found in the Pascal User Manual & Report (ref. 1). Ref. 2 is a good introduction to Pascal for those who are not familiar with programming in high-level languages.

1. Description of the existing hard- and software.

1.1. Description of the hardware configuration.

Presently, the hardware of the multiprocessor system consists of two LSI 11/23 processors (described in ref. 3) with 4k words of local memory each, and with several (CPU bound) I/O devices. The CPU's have access to a common memory of 96k words. This access is controlled by an arbiter who can grant the common memory to a CPU during one bus cycle (fig. 1).
It is possible to expand the system with more processors and more common memory blocks each controlled by a separate arbiter.
A more comprehensive description of the hardware configuration can be found in ref. 4. Ref. 5 describes the arbiter in detail.

1.2. Description of the existing software.

In the group Measurement and Control several Pascal compilers are available There is chosen to use the Ericsson compiler (for a description of the deviations from Pascal see ref. 6), because that is the one best known within the group, the source is available, and there exists already some software for the multiprocessor system, written in that language (e.g. ref. 7). Furthermore there are no important reasons to switch to another compiler.
The compiler works under the operating system RSX (ref. 8, 9, 10 et.al.). This is the reason that this operating system is chosen as development system. During execution of programs on the multiprocessor system, we can't use RSX executives (like QIO), because RSX (like all standard PDP 11 operating systems) gives no multiprocessor support. Therefore all Pascal system routines that make requests to RSX must be prohibited or modified. This applies especially to runtime error messages, because they perform I/O autonomously (i.e. without a standard procedure being called in the user program).
There is a preprocessor available, written by W.J.M. Lemmens (described in a THE report to be published). This preprocessor should be used before the Pascal compilation. Its task is, among others, the addition of the declarations of the communication and synchronisation routines, and

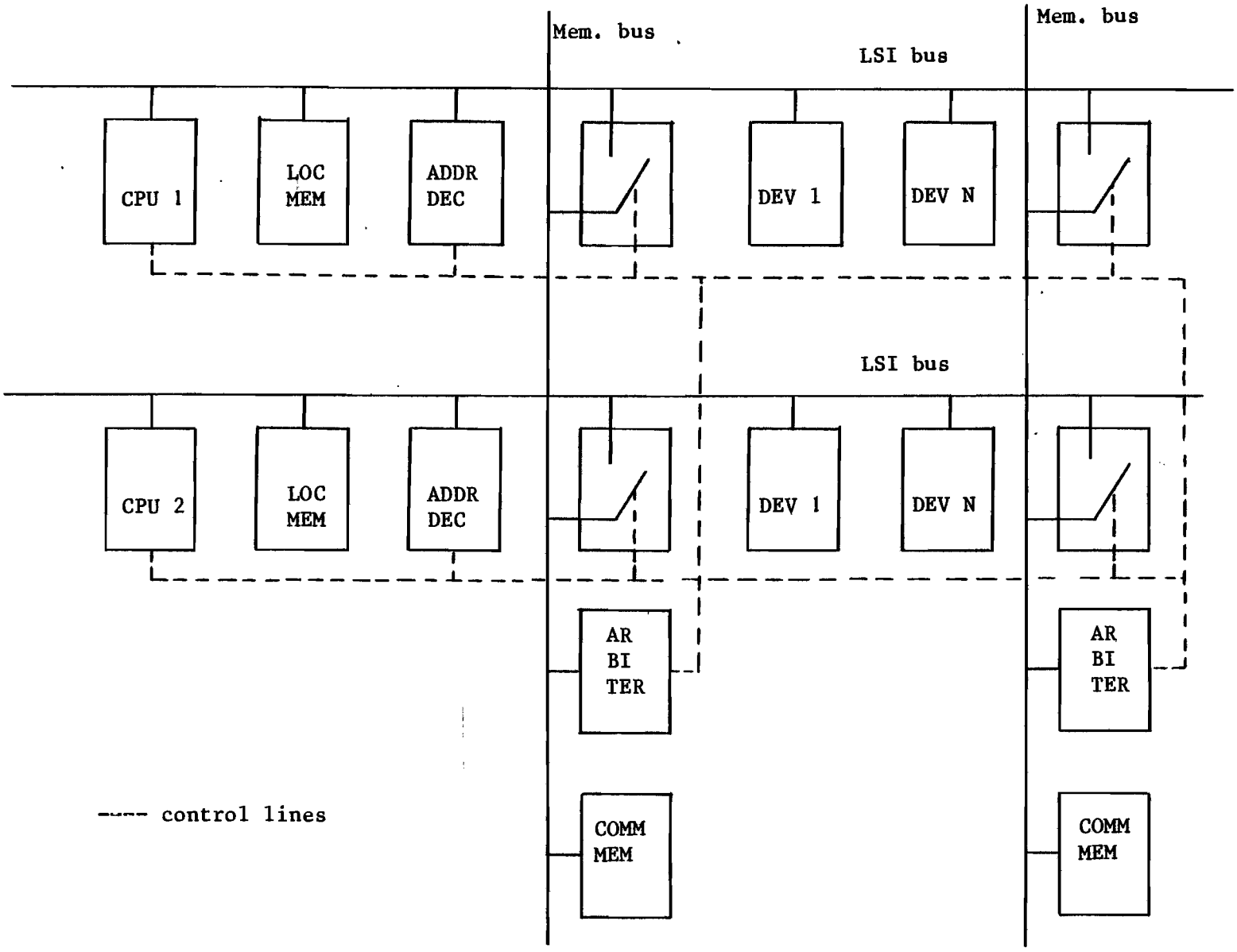


Fig. 1: Multiprocessor system.

some new standard types (e.g. semaphore). A more comprehensive description of the function of the preprocessor can be found in appendix A.

2. Problem analysis and structure of the system.

2.1. Design considerations.

The following considerations served as starting-points during the development of the software.

- As much as possible the Pascal compiler is left unaffected, because changing the compiler is a rather complicated task. This also avoids the arising of two incompatible compilers, and all problems involved with it (duplicate libraries and so on).
- The Pascal system to be developed must at least be capable to carry out all tasks, which are performed by the already existing system written in assembler (for a description see ref. 11).
- All software will, as far as possible, be written in Pascal. This will cause the system to be slower and use more memory. However, it will be easier to write, debug and document the software. Later on it is always possible to write the software (or part of it) in assembler, if this is desired.
- Procedures should preferably perform only one task.
- The system to be developed doesn't have to support the Pascal standard system routines which refer to RSX routines (e.g. read,write, get, put, etc. etc.).

2.2. General description of the structure of the system.

Software for the multiprocessor system can relatively easily be written in the form of a number of independent, co-operating, possibly cyclic processes. These processes can be executed simultaneously.
Communication and synchronisation between processes can be achieved in different ways, namely:

- by using semaphores with operations WAIT and SIGNAL.
- by using buffers with operations SEND and RECEIVE.
- by using global variables.

The number of processes, present in the system, shouldn't be limited to the number of available processors. This implies that it must be possible for a processor to serve more processes: processor sharing.

We can distinguish three states, in which a process can be in relation to the processors:

- The process is in execution on one of the processors (running).
- The process waits for execution on one of the processors (runnable).
- The process isn't ready for execution because it is waiting for data, a signal of a semaphore, access of a region or because it is completely inactive (sleeping).

A process which is runnable, waits for execution in a queue. A processor which is released (because the process it was executing has become sleeping), should continue the first process in the queue. Processes that are sleeping because they are waiting for a semaphore, a region, or data from a buffer, stay in a queue, the so-called semaphore queue, present with any semaphore, region and buffer.
Because I/O devices are CPU bound, we must distinguish between processes which do, and those which don't use an I/O device. The latter can be executed on any processor in the system, whilst the former are CPU bound. Due to this difference, there are several queues in which runnable processes must wait for execution. If a process is CPU bound it will wait in the queue of the processor it is bound to (the processorqueue). Otherwise it will wait in the readyqueue.
A processor which is released, gives priority to the processes waiting in its processorqueue. If such a process is absent, a process waiting in the readyqueue will be continued.

2.3. Alternatives to implement the processes.

To implement simultaneously running processes, written in Pascal, these fundamentally different possibilities exist:

- Processes are separate programs, which communicate through a kind of "external" variables.
- There is only one program, in which all processes are known (as procedure, or through a newly introduced PROCESS statement).

For a good consideration of the benefits and disadvantages of a choice between these alternatives, it is necessary to know how the addressing and allocation of variables in Pascal is arranged.

2.4. Addressing in Pascal.

In Pascal it is only possible to introduce new variables at the beginning of the description of the main program, or a procedure (This in contrast with for instance ALGOL, where this is possible at every block entry). In Ericsson Pascal, all variables are placed on a stack. This stack is also used to store temporary results. The structure of the stack can be seen in fig. 2.

R3 points to the begin of the software stack. (this stack grows downward.) R4 points to the place where the data of the last entered procedure begin, and R5 points to the top of the software stack. Ericsson Pascal has also some so-called hidden variables. These are placed just above the software stack, and are used by the runtime system for administration. Only a few of these hidden variables are interesting for us namely:

2(R3) which contains the linenumber of the line being executed.

4(R3) which contains the dynamic option switch word, that controls the error handling.

8.(R3) which points to the top of the heap.

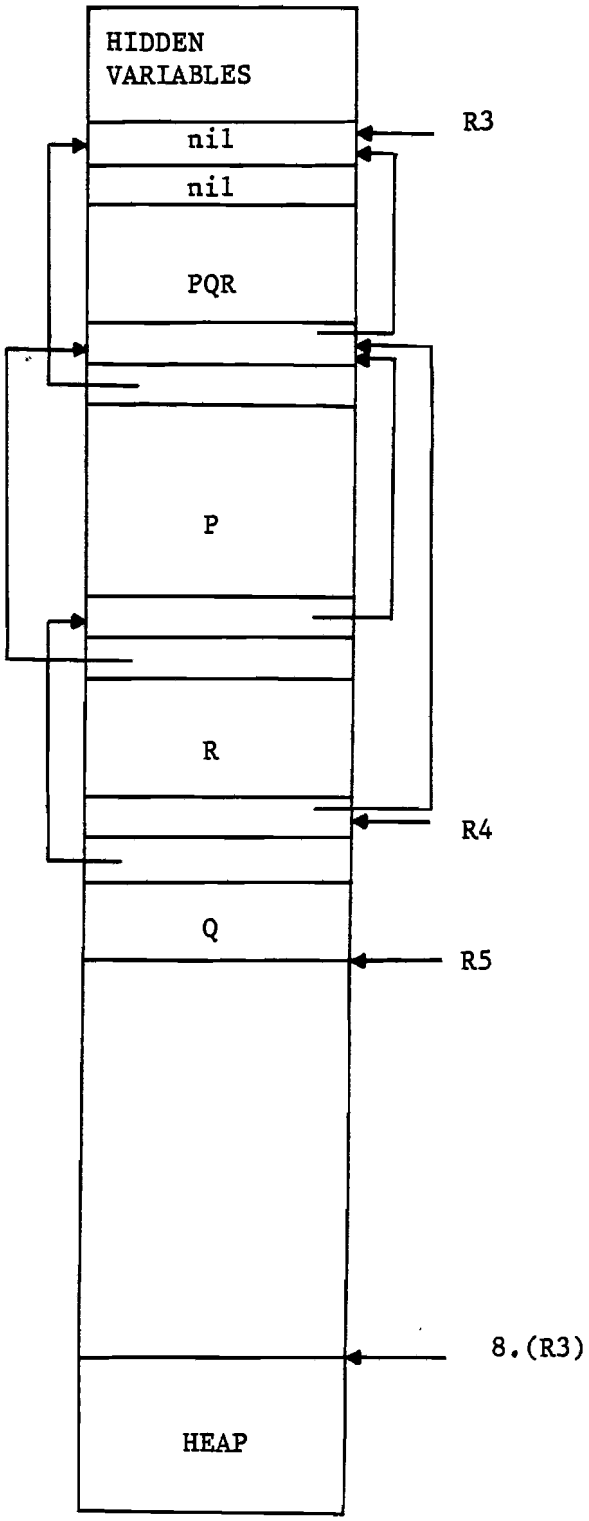
On the heap all dynamically (with NEW) created variables are placed. The heap grows upward. Global variables are always addressed via R3, local variables via R4. In all other cases there is a link, via which one can go back in the access chain. There is another chain which indicates the sequence in which procedures were called, so that, upon leaving a procedure, all the stackspace used in that procedure (e.g. from declared variables) can be released.

In the software stack, the compiler reserves space for all variables. To this space 1000. words are added. This extra space is used for:

- the heap
- temporary results
- parameter copies at procedure calls
- both chains
- "extra" variables (created by recursive procedure calls)

The software stack can be extended with an option of the RSX taskbuilder (TKB) (ref. 12).

Besides a software stack, there is also a hardware stack. Here R6 points to top of stack. The hardware stack is, among others, used to save return addresses, when calling a subroutine. The hardware stack can be extended




```
PROGRAM PQR;
    PROCEDURE P;
        PROCEDURE Q;
        BEGIN
        END;
        PROCEDURE R;
        BEGIN
            Q
        END;
    BEGIN
        R
    END;
BEGIN
    P
END.
```

Fig. 2: Example of the stack structure.
Stack after the calling sequence PQR → P → R → Q

with the TKB option STACK (ref. 12).
Further details concerning the implementation of Pascal and the addressing mechanism can be found in ref. 13, whilst ref. 14, 15, 16 and 17 describe the ETH compiler, on which the Ericsson compiler is based.

2.5. Choice of an alternative.

First lets compare the advantages and disadvantages of both alternatives to implement Pascal processes on a multiprocessor system.

1. Processes are separate programs which communicate via "external" variables.
   Problem is the creation of a common data area. This can happen via a COMMON (ref. 12). Then the software stack must be split in a common and a private part. This can only be done by changing the compiler. In the common part of the stack the common variables and the heap are placed. Also provisions have to be made to take care that all processes place a global variable with a certain name on the same address.
2. There is one program in which the statement PROCESS is known. This creates the possibility of two sub-alternatives.
   A. There is a separate stack created in the main stack for each process. This implies that we ourselves must take care for stack overflow.
   B. Every process gets an own stack. This implies a modification of the compiler, whereas NEW may also cause problems.

Alternative 2A has been chosen, because no compiler modification is needed for this, so no concession towards the design constraints has to be made. (Remark: the reserved word PROCESS is interpreted by the preprocessor.) This alternative however, has the restriction that NEW becomes a critical operation, and therefore has to be protected. Also the operations MARK and RELEASE must be forbidden, because all processes have a common heap. All the other alternatives offer no good solution for this problem either.

3. Implementation.

3.1. Implementation of the local stacks.

All process stacks are created in the software stack. This happens by claiming an amount of space from top of stack downward. The main program shall do this first, and so create a main program stack. Processes reserve their stack space when they are started. The user must indicate how much space must be reserved. For the main program this is a fixed amount (because the main program is not started by a user unlike processes). A process which is started shall first adjust its hardware and software stackpointer (R6 resp. R5), so that they point to the claimed space on the software stack. The distribution of this space between hardware and software stack must be indicated by the user when starting.

3.2. Implementation of the process descriptor.

Every process has a process descriptor, which contains several essential data of the process, namely:

- The software stackpointer (R5) (if a process is not running; when a process becomes running all registers are restored via this stackpointer).
- A pointer to the processor queue which contains the process when it is runnable.
- The processor status (if a process is, or may become, not running).
- A sequence number that identifies the process. These succesive numbers are assigned by the system during starting.
- The priority of a process. This must be supplied by the user during starting. The magnitude of the priority has influence on the place in a queue where a process is placed (see next paragraph).
- A limit for the software stackpointer, below which it may not grow, because otherwise another stack may be overwritten.
- A limit for the hardware stackpointer, below which it may not grow, because otherwise another stack may be overwritten.
- A pointer to another process descriptor. This pointer is used when the process is in a queue (see next paragraph).

3.3. Implementation of the queues.

If a process is sleeping or runnable, its process descriptor resides in a queue. Such a queue consists of a slotword (with which the exclusive right of admission to the queue is realised), and a pointer to a process descriptor. Because each process descriptor contains a pointer to a process descriptor, we can make queues as long as necessary.
If a process has to be queued, its process descriptor will be placed in the queue in such a way that all succeding processes have a lower priority, whilst all preceding processes have a higher or equal priority.
The pointer of the queue thus always points to the process descriptor with highest priority, that is placed in the queue. In cases that the highest priority is not uniquely defined, the process descriptor of the process with highest priority which is longest resident in the queue is taken as first queue element. If a process is removed from a queue, it is always the process standing in front of the queue.

3.4. Addressing the I/O page.

As a programmer of the multiprocessor Pascal system, one mostly wants to address the I/O page. (The part of the memory with physical addresses 760000-777776, which contains the device registers).
Normally one cannot address the I/O page in ones program. A user of the multisystem however, perhaps wants to do I/O. This is realized by placing a device common over the I/O page (see ref. 12 and appendix C). When building a task one can link this common into the task. This makes the I/O page (in assembler) addressable through addresses 160000-177776.

3.5. Addressing the local memory.

Because a program is loaded under RSX it can be placed anywhere in the memory (due to the normally present memory management). It is therefore not self-evident that virtual address 0 corresponds with physical address 0. (In fact this will never happen, because the lower part of the memory is used by the executive.) A multisystem programmer however wants to use the vector addresses, and perhaps the local memory too. Therefore we have

to interfere with the memory management. The memory management is described in ref. 18, p 86-88 and 255-259, and in ref. 3 chapter 9. The operation of the memory management is supposed to be known for the rest of this paragraph.
We want virtual address 0 to correspond with absolute address 0. Therefore we have to change APR 0. Problem is that normally part of the task is mapped via this APR.
Normally the first part of the task is a task header (about 274B words in size). Next in virtual memory comes the hardware stack, accessed via R6 (SP). Usually the size of this stack is 1k words. The size can be extended with the TKB option STACK. The header is not necessary when we have left RSX, so it may be overwritten. If we now enlarge the stack with say 4k words (so part of the stack is mapped via APR 1), we can change APR 0, so that it points to the lowest 4k words (PAR 0 = 0, PDR 0 = 77406B). This makes the lowest 4k words addressable via addresses 0 - 177777.
The structure of the local memory can be found in appendix D.

4. Error handling.

4.1. Detecting an error.

Most of the errors which can occur, are detected by the Pascal runtime system. The implemented extensions involve some new error causes.

- process hardware stack overflow
- process software stack overflow
- heap overruns process stack or new started process overlaps heap
- in STRINGTOINTEGER one tries to convert a string containing illegal characters
- the string to be converted in STRINGTOINTEGER doesn't fit in the integer (overflow)
- the integer to be converted in INTEGERTOSTRING doesn't fit in the string (too small)

The third error can only appear after a call of NEW, or after the creation of new process stacks. Every time this error can arise, the error conditions are checked, and, if necessary, an error message is generated.
Overflow of the hardware or software stack can occur at any moment. Because testing on overflow after each statement execution gives a lot of overhead, it is implemented as follows: (analogous to Ericsson Pascal)
At procedure entry is tested if a certain amount of space is available on both stacks, and if not, an error message is generated.
The minimal required stack space is 40. words on the software stack and 10. words on the hardware stack. This is completely conform Ericsson Pascal.
The last three errors can only occur in the procedures mentioned, and are detected there.

4.2. Generating an error message.

Normally, the error messages generated by Pascal are handed over to RSX, which prints them. This is impossible in our system, (because we don't have RSX at our disposal). Therefore all error messages are send to the standard buffer ERRORBUF. There is also a standard process ERRORLOG available, which converts messages from ERRORBUF to characters, which

thereafter can be printed by the standard process TTUIT (see for a description of the standard processes appendix E).
A problem that arises when generating error messages concerns the linenumbers. In the error message the number of the line in which the error occurs, is given. This linenumber is stored in one of the hidden variables (see paragraph 2.4.). If, on our multiprocessor system, more processes are executed simultaneously, this hidden variable should contain more linenumbers, which of course is impossible. To avoid this a small compiler modification is made (see appendix F), so that linenumbers are saved in a location in the local memory, instead of the hidden variable. This hidden variable is now used to store the address of ERRORBUF.

5. Limitations and suggestions for extensions of the system.

The implemented system has some limitations in it. The known ones will be described here shortly. A possible solution is also given.

- The total amount of memory which can be used by the system is limited to 24k words (+ 4k local memory + 4k I/O page).
  This is because (at the moment) there exists no possibility for overlaying. Perhaps this can be realized with the already designed file system (ref. 19).
- It is impossible to delete processes completely (and return the used memory to the system). This could possibly be solved by a good overlaying mechanism as well.
- It is impossible to change the size of the process stacks dynamically. Again overlaying could realize this feature.
- Some more standard routines could be implemented (e.g. read, write, date, time).
- Programming could be simplified by implementing a multiprocessor version of the symbolic debugger.
- A PAP compiler could speed up and simplify the compiling and task-building. Also the implementation of the standard types can be hidden for the user.
- The system is slow compared to the system written in assembler. This is caused by the overhead and inefficiency of Pascal. Speed can be increased by writing the system procedures in assembler or microcode instead of Pascal.
- Processes which have executed a BYE (see appendix E) can never be restarted. This can easily be changed, but then it is necessary that processes can be identified by their names (for instance this name can be placed in the process descriptor).
  Also a new procedure has to be created to allow processes to leave the system without restart possibility. (This can be used to exit from noncyclic processes).
- The execution speed of user processes can be enlarged by adding more arbiters and processors. A description of the system changes caused by hardware changes is given in appendix G.

Conclusions.

The system developed is in quite good agreement with the design considerations mentioned in paragraph 2.1. On some points however, some tedious limitations exist. These limit the usability of the system.
The system is rather flexible, modular and easy to maintain. A disadvantage is its dependence on Ericsson Pascal, especially on the stack structure, the addressing mechanism, and the property of the compiler not to save the registers upon external procedure entry.
Also, when wanting to use another compiler, the source must be available, in order that the compiler modification can be made.

References.

1. K. Jensen, N. Wirth: Pascal User Manual & Report, Springer Verlag, 1975.
2. J. Welsh, J. Elder: Introduction to Pascal, Prentice Hall, 1979.
3. Microcomputer processor handbook, Digital Equipment corp., Maynard Mass., 1980.
4. M.J. Kanters: Hardware for a "Multiprocessor system" with LSI-11. In: Proceedings of the Digital Equipment Computer Users Society, volume I: Europe, 1979.
5. P.J.J.M. Mostert: Het vervangen van de LSI 11/03 CPU door een LSI 11/23 CPU in het multiprocessorsysteem van de vakgroep ER, stageverslag THE, Eindhoven, 1981.
6. S. Torstendahl, Pascal User Manual, LM Ericsson.
7. F.J.H.M. Meulenbroeks: Sequential Pascal met synchronisatieprimitieven als real-time taal voor parallelle processen, stageverslag THE, Eindhoven, 1980.
8. Introduction to RSX-11 M, Digital Equipment corp., Maynard Mass.
9. RSX-11 M/M-plus MCR operations manual, Digital Equipment corp., Maynard Mass., 1979.
10. RSX-11 Utilities Manual, Digital Equipment corp., Maynard Mass., 1979.
11. W.P.M. den Dekker: Ontwikkeling van een software pakket voor een multiprocessor systeem, stageverslag THE, Eindhoven, 1979.
12. RSX-11 M/M-plus Task builder manual, Digital Equipment corp., Maynard Mass.,1979.
13. W. de Vries: An implementation of the language Pascal for the PDP 11 series, based on a portable Pascal compiler, Verslag doctoraalopdracht, THT, 1975.
14. U. Ammann: Die Entwicklung eines Pascal Compilers nach der Methode des Structurierten Programmieren, Diss. ETH 5456, Zürich, 1976.
15. U. Ammann: On code generation in a Pascal compiler, ETH Zürich, 1976.
16. U. Ammann: The Zürich Implementation. In: Pascal - The language and its Implementation, D.W. Barron (editor), J. Wiley, 1981. p. 63-82.
17. U. Ammann: Code generation for a Pascal Compiler. In: Pascal - The Language and its Implementation, D.W. Barron (editor), J. Wiley, 1981, p. 83-124.

18. R.H. Eckhouse, L.R. Morris: Minicomputer systems, Prentice Hall, 1979.
19. H.J.M. van Kruysdijk: Een filesysteem voor een multiprocessor-configuratie, afstudeerverslag THE, Eindhoven, 1981.
20. F.J. Kylstra, Procescomputersystemen, dictaat bij het college nr. 5.335.0, THE, Eindhoven, 1979.

Appendix A: Description of the function of the preprocessor.

The preprocessor transforms a program source, written in Pascal-plus (PAP) (User manual see appendix B), into a file which can be compiled by the Ericsson Pascal compiler. The operational characteristics of the preprocessor are:

- addition of the declarations of the PAP standard and system types: WORD, ADDRESS, PDPT, QPOINTER, PRODESC, QUEUE, SEMAF, REGION, MESBUF, ERRORMES, ERRORBUFTYPE.
- addition of the declaration of the standard and system variables: PDPOINTER, R5, PROCAQ, PROCBQ, READYQ, BYEQUEUE, NUMBER, TEMPQPNTR, TOPOFSTACK, NEWREG, STARTREG, ERRORBUF.
- addition of the declaration of the standard and system procedures, functions and processes: INITIALIZE, PHOENIX, SIGNAL, WAIT, ATTACH, INIT, BYE, SEND, RECEIVE, STRINGTOINTEGER, INTEGERTOSTRING, READREG, WRITEREG, NEWSTK, OVFLCHK, NEWCHK, INITTOS, TTIN, TTUIT, ERRORLOG.
- replacing the word PROCESS with PROCEDURE.
- inserting a call of INIT as first process statement.
- checking the type compatibility of messages and buffers in SEND and RECEIVE calls.
- inserting PDPOINTER as the first parameter of procedures declared as SYSTEM procedures (or functions).
- replacing of the word CYCLE with WHILE TRUE DO BEGIN
- replacing of the word REGENTER with BEGIN WAIT( );
- replacing of the word REGEXIT with ;SIGNAL( ) END
- checking if regions are nested correct.
- placing a region around each NEW statement.
- if no CYCLE occurs: addition of a call of BYE at the end of each process.
- within each process: inserting the declaration of a local variable PDPOINTER.
- inserting a call of OVFLCHK as first statement of each procedure.
- inserting a call of NEWCHK after each call of NEW.
- replacing the user declaration MSGBUF n OF with the appropriate MSGBUF declaration, and initializing this buffer.
- replacing the user declaration MESSAGE n OF with ARRAY [0..n-1] OF

- inserting some statements to provide the main program with a process stack.
- replacing the user call of START with a call of NEWSTK followed by a call of the process. Around these calls a region is placed.

A more comprehensive description of the preprocessor is to be given in a TH report of W.J.M. Lemmens, which is under preparation.
A description of the purpose of the various types, variables, procedures and functions can be found in appendix E.

Appendix B: Pascal-plus user manual.

PASCAL-PLUS USER MANUAL

Summary.

This manual describes the extensions and limitations of Pascal-plus (PAP) compared with Pascal.

Contents.

Summary	1
Contents	1
Introduction	2
Process	3
Semaphores and regions	4
Messages and buffers	5
Error handling	6
Auxiliary types, routines and statements	7
An implementation	8
How to use Pascal-plus	9

Introduction.

Pascal-plus (PAP) is a language based on Pascal (described in (1)). PAP is intended as a programming language for parallel processes. For this purpose the language Pascal is extended with some communication and synchronisation primitives. This manual describes the extensions of PAP as compared to Pascal. Also some system dependent features are described. These are all based on Ericsson Pascal (2), a dialect of Pascal.

1. K. Jensen, N. Wirth: Pascal User Manual & Report, Springer Verlag, 1975.
2. S. Torstendahl: Pascal User Manual, LM Ericsson.

Process.

To admit parallel processes, PAP is extended with the special symbol PROCESS. Processes are treated the same way as procedures (so can have parameter lists etc.). It is however not allowed to call processes.
A process must be started with a START statement. This statement is of the form:

START(QUEUE,PRI,HL,SL,PROCESS NAME {,process parameters})

where { } means: zero or more times the enclosed.
Parameters are:

QUEUE: the processor queue of the process (especially for multiprocessor systems. The queues have (conventionally) the names
READYQ (queue of processes which can be executed by any processor)
PROCAQ (queue of the first processor)
PROCBQ (queue of the second processor)
and so on

PRI : priority of the process

HL : hardware stacksize for the process

SL : software stacksize for the process

PROCESS NAME: name of the process to be started

and parameters conform the process heading

Processes can remove themselves by executing the BYE statement.
The following standard processes are already predeclared:

PROCESS ERRORLOG(CHARBUF)

This process receives error messages from the predeclared buffer ERRORBUF, and places them character after character in CHARBUF. CHARBUF must be a buffer for messages of the type ASCII or BYTE.

PROCESS TTIN(CHARBUF)

This process receives characters from a terminal and places them in CHARBUF. The terminal must be on addresses 177560/177562 (status/data registers) and vectored via addresses 60/62.

PROCESS TTUIT(CHARBUF)

This process receives characters from CHARBUF and sends them to a terminal. The terminal must be on addresses 177564/177566 (status/data registers) and vectored via addresses 64/66.

Semaphores and regions.

For synchronisation purposes PAP is extended with the type SEMAF, representing a semaphore. On this type three operations exist:

PROCEDURE WAIT(VAR SEM:SEMAF)

delays the calling process if the semaphore counter is zero, otherwise decrements this counter by one.

PROCEDURE SIGNAL(VAR SEM:SEMAF)

If no process is delayed in the semaphore queue the semaphore counter is incremented, otherwise the process with highest priority is placed in its processor queue. Processes with equal priority are awoken on a first come first served basis.

PROCEDURE ATTACH(DEV:ADDRESS;VAR SEM:SEMAF)

An interrupt service routine is generated, which couples a signal on semaphore SEM to an interrupt vectored via address DEV. ATTACH doesn't enable or disable interrupts.

Also the type REGION is implemented. This type can be used to protect critical sections. A critical section can be made with a REGENTER statement. This statement is of the form

REGENTER region variable statement {;statement} REGEXIT region variable

where region variable is a variable of the type REGION.

Only one process at a time can be in a REGENTER statement with a certain region name.

Warning: Nested REGENTER statements with the same region variable create a deadlock!

Messages and buffers.

For communication between processes PAP is extended with the types MESSAGE and MSGBUF. Variables of the MESSAGE type must be declared as

MESSAGE nrofelements OF elementtype

where elementtype may be INTEGER, CHAR, ASCII or BYTE, and nrofelements represents the number of elements of the message. Variables declared as MESSAGE can be treated as ARRAY [0.. nrofelements-1] OF elementtype. Variables of the MSGBUF type must be declared as

MSGBUF nrofelements OF messagetype

nrofelements is the number of messages that can be placed in the buffer of the MSGBUF type.

On messages and buffers two procedures operate:

PROCEDURE SEND(MESSAGE,BUFFER)

PROCEDURE RECEIVE(MESSAGE,BUFFER)

which place resp. receive the message in resp. from the buffer. The types of the message and the buffer elements must match.

Error handling.

All runtime error messages are placed in the standard buffer ERRORBUF, which is of the type ERRORBUFTYPE. This is a buffer which can contain 10 messages of the type ERRORMES (array [0..29] OF BYTE). Errors can be printed by declaring a character buffer and starting the processes ERRORLOG and TTUIT with this buffer as parameter. Preferably these processes should be started first.
Some new error messages are implemented, namely:

90 (fatal) : stack overruns heap or vice versa
Too many NEW statements or too much stackspace reserved in START statements.

91 (fatal) : local hardware stack overflow (10. words left).

92 (fatal) : local software stack overflow (40. words left).

94 (warning): illegal character in string parameter of STRINGTOINTEGER. zero taken as conversion result. Error message displays the offending value.

95 (warning): overflow in STRINGTOINTEGER. MAXINT or MININT taken (dependent on sign)

96 (warning): string in INTEGERTOSTRING too small; filled with blanks.

A fatal error results in a BYE for the offending process, unless the H-switch is set to "continue after fatal error". Fatal error 90 also stops the process which executes the START statement.

Auxiliary types, routines and statements.

Some types, routines and statements are added to PAP for convenience of the user. The types are ADDRESS and WORD, which can be treated as integer. Two routines for addressing virtual memory locations are available:

```
PROCEDURE WRITEREG(PART:INTEGER;REG:ADDRESS;CONT:WORD)
FUNCTION READREG(PART:INTEGER;REG:ADDRESS):WORD
```

These can be used to write or read data resp. to or from a virtual address. In case of write CONT gives the data to be written.
The virtual address is split in two parts* and constructed with the formula

$$PART * 2^{12} + REG$$

or in words: PART gives the highest 3 bits of the address and REG the lowest 13.
The following two routines are written to allow conversion from strings to integers and vice versa.

```
PROCEDURE INTEGERTOSTRING(I:INTEGER;STRING S)
PROCEDURE STRINGTOINTEGER(STRING S;VAR I:INTEGER)
```

INTEGERTOSTRING converts an integer to a string. The converted integer is placed in the string with preceding blanks, if necessary. A sign is always placed in the string. If the string can't contain the converted integer an error message is generated.
STRINGTOINTEGER converts a string to an integer. Blanks in the string are ignored, only a sign and digits are allowed. For all other characters in the string an error message is generated. In case of overflow an error message is generated too.
Both procedures use the string parameter implementation of Ericsson Pascal.
PAP is extended with the CYCLE statement to allow processes to repeat themselves infinitely. The statement is of the form:

```
CYCLE statement {;statement} END
```

The statements in the CYCLE statement are repeated infinitely. The CYCLE statement (if used) must be the last statement of a process declaration. Also the CYCLE statement may not occur in other statements (e.g. IF statements).

* This is done because integers (15 bit + sign bit) can't be interpreted as addresses (16 bit) easily.

An implementation.

PAP is implemented on a multiprocessor system with two LSI 11/23 's with shared memory. The implementation is based on the Ericsson Pascal compiler. All previously described procedures are made for this configuration. The implementation is however not conform the previous descriptions. The following modifications are made:

- Structured types can't have messages, buffers or semaphores as elements. Arrays with these elements can be declared with the type VECTOR, which is of the form:

  VECTOR [lb..hb] OF elementtype

  where elementtype can be semaphore, message or buffer.

  A vector can be treated as an array with lower bound lb and upper bound ub.
- Type checking for procedures, functions and processes is suppressed when the type of a procedure, function or process parameter is preceded by the special symbol UNIV. Use of this feature is not recommanded for normal use.
- Procedure and function declarations can be preceded by the special symbol SYSTEM. This provides a possibility to identify the calling process. Its implementation and consequences are not discussed here.

How to use Pascal-plus.

A file, written in Pascal-plus, can be transformed into a taskfile in the following way:
First transform the program source to a Pascal source. This can be done by the preprocessor, which can be activated with

PAP <passource>,<list>=<papsource>

If no file designators are given, the preprocessor will prompt with PAP>, and then the user can give the file specifications.
Default extensions are .PAS, .LST and .PAP respectively. Default device is SY: and default uic is the uic under which PAP is run.
The passource and listfiles may be omitted.
If no error occurs, the passource file can be compiled with the Ericsson Pascal compiler. See, for a detailed description of it, the User Manual.
Taskbuilding the file is also conform the description in the Ericsson Pascal User Manual. A user must however enter the options:

STACK = 4000

and

COMMON = IONEW:RW

Furthermore the option EXTTSK must be given a value, greater than the total stackspace used by all processes (the main program takes 330. words stackspace), plus all global declared variables plus all variables created with NEW during program execution.
The now created task can be executed.

Appendix C: Generating a device common.

We want to create a device common, with which we can address the whole I/O page in a program. This can be done as follows:
First create a data area of 4k words. This can be realized with the following program (IONEW.MAC):

```
.TITLE  IONEW
.PSECT  IONEW
.BLKW   4095.
.END
```

This program is translated with the MACRO compiler:

```
MAC IONEW=IONEW
```

Taskbuilding of the common goes as follows:

```
TKB
TKB> IONEW/-HD/PI,IONEW,IONEW=IONEW
TKB> /
ENTER OPTIONS:
TKB> STACK=0
TKB> PAR=IONEW:0:20000
TKB> //
```

(underlined parts are RSX responses)
/PI means: the common is position independent
/-HD means: give the task file no header
STACK=0 means: we don't want a stack
and PAR=IONEW:0:20000 means that the program must be loaded into partition IONEW, starting at virtual address 0, and that the partition size is 20000.
The common is now created. If we want the common to be placed over the I/O page, we must create a partition IONEW over the I/O page, and fill this partition with our common. Creating a partition can be done with the (privileged) MCR command (ref. 9):

```
SET /MAIN=IONEW:7600:200:DEV
```

The created partition can be initialized by running the task made by TKB (IONEW.TSK). One can address the device common in a program if one gives the taskbuilder as option

RESCOM=IONEW/RW

or COMMON=IONEW:RW

dependent on whether the file IONEW.STB (made by the taskbuilder) resides on the own account or on [1,1] . The letters RW denote that one wants to use the common for both read and write.

Appendix D: Structure of the local memory.

The local memory can be divided into three parts:

- addresses 0-777

  These addresses are used for interrupt vectors. Most of it however is not used. All current devices have a vector address < 377.

- addresses 772-1220

  These addresses are used by the runtime system. They can be subdivided as follows:

  772 : contains the linenumber of the line in execution.

  774 : contains the address of the processorqueue of that processor.

  776 : contains the pdpointer of the process currently being executed.

  1040-1000: Kernel hardware stack
  Used during interrupt service routines, created with ATTACH, and during traps, caused by an error.

  1100-1040: User hardware stack
  Used during "idling" (when trying to get a process from a queue for execution).

  1150-1100: software stack during idling.

  1220-1150: software stack during interrupt service routines.

- addresses 1220-17777

  Free for user applications and future system extensions.

Appendix E: Software description.

Introduction.	E-2
New types.	E-3
New variables.	E-7
New runtime errors.	E-9
Description of the procedures and functions.	E-10
Description of the standard processes.	E-49

Introduction.

This appendix gives the implementation details of the system. All new types and variables are described. Also a description of the procedures, functions and system processes is given. These are however not described at statement level. Readers interested in all fine implementation details are referred to the listings, and the comments inserted there.

New types.

The following types are added to Pascal to create PAP, or as hulptypes for the implementation of the system.

```
PDPT = ^PRODESC
```

defines a pointer to the type PRODESC

```
QPOINTER = ^QUEUE
```

defines a pointer to the type QUEUE

```
PRODESC = RECORD
            SP:WORD;
            PROCXQ:QPOINTER;
            PS:INTEGER;
            NR:INTEGER;
            PRI:INTEGER;
            HST:INTEGER;
            SST:INTEGER;
            NEXT:PDPT
          END
```

defines a process descriptor. A process descriptor contains some vital information of a process.
The fields have the following meaning:

SP : contains the software stackpointer (R5) of a sleeping process.
PROCXQ: pointer to the queue in which a process resides when it is runnable.
PS : contains the processor status of a sleeping process.
NR : contains a sequence number, assigned when a process is started. The first started process has NR = 0.
HST : contains the limit which may not be exceeded by the hardware stackpointer (R6) when a procedure is entered.
SST : contains the limit which may not be exceeded by the software stackpointer (R5) when a procedure is entered.
NEXT : pointer to another process descriptor. This field is, among others, used to implement queues.

```
WORD = INTEGER
     a hulptype

ADDRESS = WORD
     a hulptype

QUEUE = RECORD
          SLOT:WORD;
          GET:PDPT
        END
```

defines a queue in which a process can reside when sleeping or runnable. The fields have the following meaning:

SLOT: used for gaining access to the queue.
SLOT = 0 means: a process is operating on the queue
SLOT = 1 means: no process is operating on the queue
initial value: SLOT = 1

GET : points to the first process descriptor in the queue.

```
SEMAF = RECORD
          SLOT:WORD;
          CTR:INTEGER;
          GET:PDPT
        END
```

defines a semaphore. The fields mean:

SLOT: see under QUEUE

CTR : the semaphore counter; initially 0

GET : see under QUEUE

```
REGION = SEMAF
```

defines a region. The implementation is analogue to SEMAF, the function is different. REGION variables have initially CTR = 1

```
MSGBUF = RECORD
           EMPTY:SEMAF;
           FULL:SEMAF;
           MUTEX:SEMAF;
           LENGTH:INTEGER;
           NBYT:INTEGER;
           GETPT:INTEGER;
           PUTPT:INTEGER;
           BUF:ARRAY [0..1] OF ARRAY [0..1] OF BYTE
         END
```

defines a buffer, to be used in the declarations of SEND and RECEIVE. The fields have the following meaning:

EMPTY : Semaphore whose counter gives the number of free places in the buffer.

FULL : Semaphore whose counter gives the number of full places in the buffer.

MUTEX : Semaphore used to create a mutual exclusion for the buffer access.

LENGTH: holds the number of messages that fits in the buffer

NBYT : holds the width of a message for the buffer (in bytes)

GETPT : pointer to the first filled bufferplace

PUTPT : pointer to the first free bufferplace

BUF : the buffer itself. The type is used in the declarations of SEND and RECEIVE with the reserved word UNIV, which suppresses typechecking, so the bounds of BUF have no meaning.

```
ERRORBUFTYPE = RECORD
                 EMPTY:SEMAF;
                 FULL:SEMAF;
                 MUTEX:SEMAF;
                 LENGTH:INTEGER;
                 NBYT:INTEGER;
                 GETPT:INTEGER;
                 PUTPT:INTEGER;
                 BUF:ARRAY [0..9] OF ERRORMES
               END
```

defines the type of an errorbuffer. All fields have the same meaning as in MESBUF, except that BUF defines a buffer of 10 error messages (type ERRORMES)

ERRORMES = ARRAY [0..29] OF CHAR

defines an error message. These have the following format:

PAPRUN -- ERROR nn mmmmm vvvvv

where nn = error number

mmmmm = linenumber where the error occurred (not correct if the error occurs in a part of the program where the runtime check option R is switched off)

vvvvv = offending value (error 12 and 94 only)

New variables.

Some variables are needed for implementation purposes. Here a description of these variables is given.

PDPOINTER:PDPT

This variable is added to each process and to the main program. It is used as a pointer to the process descriptor of the process, in which it is declared. This gives a process the possibility to refer to its own process descriptor.

R5:INTEGER

Used to store the value of register 5 temporary when another process is started.

PROCAQ,PROCBQ,READYQ,BYEQUEUE:QPOINTER

Pointers to the queue of processor A, processor B, the readyqueue (which contains the processes that can be executed by any processor), and the byequeue (where all processes are placed that have executed a BYE).

NUMBER:INTEGER

Contains the number of processes started. Used to initialize the process descriptor field NR.

TEMPPDPT:PDPT

Temporary pdpointer used when creating a new process descriptor.

TOPOFSTACK:INTEGER

Points to the top of the space claimed on the software stack. Used when generating new stacks, and during error checking.

NEWREG:REGION

Region to protect the hidden variable 8.(R3) which points to the top of the heap.

STARTREG:REGION

Region to protect the variables R5 and TEMPPDPT when starting a process.

ERRORBUF:ERRORBUFTYPE

The (standard) buffer in which all error messages are placed.

New runtime errors.

A few new runtime errors are implemented. Their meaning is explained below.

Error 90: Fatal

This error means that the heap and the software stack overrun each-other. Too many variables are created with NEW, or too much stack space is claimed for process stacks.

Error 91: Fatal

This error means that a hardware stack overflow can occur. Usually caused by too many recursive procedure calls.

Error 92: Fatal

This error means that a local software stack overflow can occur.

Error 94: Warning

This error means that during the conversion of a string to an integer (in STRINGTOINTEGER) an illegal character is found.(no figure, sign or blank). The ascii value of the character is given.

Error 95: Warning

This error means that during the conversion of a string to an integer (in STRINGTOINTEGER) overflow occured. This integer is set to MAXINT or MININT (depends on the sign).

Error 96: Warning

This error means that the string given as parameter to INTEGERTOSTRING cannot contain the converted integer + sign. The string is filled with blanks.

Description of the procedures and functions.

The next pages all give a description of a procedure or function.
The following data are given:

- Heading
- Task of the procedure/function
- Parameter description
- Called procedures and functions
- Calling procedures and functions.
- Suggestions for extensions
- Further remarks

Under "Called procedures and functions" no standard functions are mentioned.

Description of the procedure ATTACH.

Heading

PROCEDURE ATTACH(DEV:ADDRESS;VAR SEM:SEMAF)

Task of the procedure

attach a signal on a semaphore to an interrupt vector

Parameter description

DEV: address of the interrupt vector

SEM: the semaphore that must be signalled by an interrupt

Called procedures and functions

ATTMAC

Calling procedures and functions

none

Suggestions for extensions

Limit the value of DEV to 777B max. (No higher interrupt vectors exist)

Further remarks

The space for the interrupt service routine is created on the heap by NEW.

Description of the procedure ATTMAC.

Heading

PROCEDURE ATTMAC(DEV:ADDRESS;VAR SEM:SEMAF;BLOK:ATTBLK)

Task of the procedure

Generating an interrupt service routine

Parameter description

DEV : address of the interrupt vector

SEM : the semaphore to be attached to an interrupt

BLOK: beginaddress of the interruptroutine

Called procedures an functions

SIGNAL (in the interrupt service routine)

Calling procedures and functions

ATTACH

Suggestions for extensions

none

Further remarks

The interrupt service routine signals the given semaphore.

Interrupt service routines run on processor priority 7.

Description of the procedure BSARPS.

Heading

PROCEDURE BSARPS(VAR SLOT:WORD;PS:INTEGER)

Task of the procedure

executing a busy signal on the variable slot and restoring the processor status

Prameter description

SLOT: word, which controls access to a queue

PS : contains the processor status, the calling process should get

Called procedures and functions

BSIGNL

RESPS

Calling procedures and functions

SIGNAL

QLEAVE

CHANGE

WAIT

QENTER

Suggestions for extensions

none

Further remarks

BSARPS means Busy Signal And Restore Processor Status

Description of the procedure BSIGNL.

Heading

PROCEDURE BSIGNL(VAR SLOT:WORD)

Task of the procedure

Move a 1 to SLOT in an indivisible operation

Parameter description

SLOT: the variable to be filled with a 1

Called procedures and functions

none

Calling procedures and functions

BSARPS

Suggestions for extensions

none

Further remarks

none

Description of the procedure BWAIT.

Heading

PROCEDURE BWAIT(VAR SLOT:WORD)

Task of the procedure

Making the processor priority 7
Waiting until there is a 1 in the slotword, and make it zero
(in an indivisible action)

Parameter description

SLOT: variable which must be made zero

Called procedures and functions

none

Calling procedures and functions

SPSABW

Suggestions for extensions

none

Further remarks

none

Description of the procedure BYE.

Heading

PROECEDURE BYE(VAR PDPOINTER:PDPT)

Task of the procedure

Stop the execution of the calling process

Parameter description

PDPOINTER: Process descriptor of the calling process

Called procedures and functions

SPSABW

CHANGE

Calling procedures and functions

$WRERROR

Suggestions for extensions

Create the possibility to restart a process that has executed a BYE

Further remarks

The procedure places the process in the BYEQUEUE, from which processes never awake.

Description of the procedure CHANGE.

Heading

PROCEDURE CHANGE(VAR SLOT:WORD;VAR GET:PDPT)

Task of the procedure

Place the calling process in a queue, release the queue, and wake a new process.

Parameter description

SLOT: word, which controls access to the queue in which the calling process must be placed.

GET : pointer to the first process descriptor of the queue, in which the calling process must be placed.

Called procedures and functions

QIN

BSARPS

WAKEUP

Calling procedures and functions

WAIT

BYE

Suggestions for extensions

none

Further remarks

none

Description of the procedure CHREG.

Heading

```
PROCEDURE CHREG(TEMPPDPT:PDPT;HSBEG:INTEGER;SSBEG:INTEGER;VAR R5:INTEGER)
```

Task of the procedure

- saving of R4, R5, R6, the processor status, the pdpointer and the linenumber on the software stack
- saving register 5 in variable R5
- initializing of R5 and R6
  R5 := SSBEG
  R6 := HSBEG
- initializing the just created new stack
  (7 words on the software stack, 2 words on the hardware stack)

Parameter description

TEMPPDPT: address of the process descriptor of the process to be started
HSBEG : initial value of the hardware stackpointer
SSBEG : initial value of the software stackpointer
R5 : variable for saving R5 of the caller

Called procedures and functions

none

Calling procedures and functions

NEWSTK

Suggestions for extensions

check if stacks aren't made too small (if a minimal size is required)

Further remarks

none

Description of the procedure COPYFROM.

Heading

```
PROCEDURE COPYFROM(VAR FROM:CHAR;VAR TOT:CHAR;OFFSET:INTEGER;
                                                NROFBYTES:INTEGER)
```

Task of the procedure

Copying nrofbytes bytes from address from + offset to the address tot

Parameter description

FROM : address that, when offset is added, contains the first byte to copy

TOT : address where the first byte is copied to

OFFSET : hulp variable to create the address of the first byte to copy

NROFBYTES: contains how many bytes must be copied

Called procedures and functions

none

Calling procedures and functions

RECEIVE

Suggestions for extensions

none

Further remarks

none

Description of the procedure COPYTO.

Heading

```
PROCEDURE COPYTO(VAR FROM:CHAR;VAR TOT:CHAR;OFFSET;INTEGER;
                                              NROFBYTES:INTEGER)
```

Task of the procedure

Copying nrofbytes bytes from address from to address tot+offset

Parameter description

FROM	: address of the first byte to copy
TOT	: address where, when offset is added, the first byte is copied to
OFFSET	: hulp variable to create the address where the first byte must be copied to
NROFBYTES:	contains how many bytes must be copied

Called procedures and functions

none

Calling procedures and functions

SEND

Suggestions for extensions

none

Further remarks

none

Description of the procedure IEBW.

Heading

PROCEDURE IEBW(VAR ERRORBUF:ERRORBUFTYPE)

Task of the procedure

Initializing 2(R3) with the address of ERRORBUF

Parameter description

ERRORBUF: buffer where all errormessages are send to

Called procedures and functions

none

Calling procedures and functions

INITIALIZE

Suggestions for extensions

none

Further remarks

In Ericsson Pascal the hidden variable 2(R3) is used for storing the linenumber. This is now stored in the local memory.
IEBW means Initialize ErrorBuffer Word.

Description of the procedure INIT.

Heading

PROCEDURE INIT(VAR PDPOINTER:PDPT)

Task of the procedure

Placing the calling process in its processor queue.

Returning to the program part that started the calling process

Parameter description

PDPOINTER: pointer to the process descriptor of the calling process

Called procedures and functions

RETTOM

OVFLCHK

Calling procedures and functions

a just started process

Suggestions for extensions

none

Further remarks

none

Description of the procedure INITIALIZE.

Heading

PROCEDURE INITIALIZE

Task of the procedure

Initialization of the processor queue's, byequeue, readyqueue, number, startreg, newreg and errorbuf.
Placing the address of errorbuf on 2(R3)

Parameter description

none

Called procedures and functions

IEBW

Calling procedures and functions

main program when started

Suggestions for extensions

none

Further remarks

none

Description of the function INITTOS.

Heading

FUNCTION INITTOS:INTEGER

Task of the function

Displaying the value of R5 (in the function result)

Parameter description

INITTOS: contains after execution the value of R5

Called procedures and functions

none

Calling procedures and functions

none

Suggestions for extensions

none

Further remarks

Function call is added by PAP to initialize the variable TOPOFSTACK.

Description of the procedure INTEGERTOSTRING.

Heading

PROCEDURE INTEGERTOSTRING(I:INTEGER;STRING S)

Task of the procedure

Converting an integer to a string (array of CHAR or part of it)

Parameter description

I: integer to be converted

S: conversion result

Called procedures and funcions

SIMPLEERROR

Calling procedures and functions

none

Suggestions for extensions

none

Further remarks

The conversion result is always preceded by a sign. If there is more space in the string than needed the conversion result is filled with preceding blanks.

If the string cannot contain the converted integer + sign an error message 96 is generated. The string is filled with blanks.

Description of the procedure NEWCHK.

Heading

PROCEDURE NEWCHK(TOPOFSTACK:INTEGER)

Task of the procedure

Testing if the software stack and the heap overrun eachother

Parameter description

TOPOFSTACK:contains the address which represents the top of the software stack

Called procedures and functions

$WRERROR

Calling procedures and functions

NEWSTK

Suggestions for extensions

none

Further remarks

If an error occurs an error message 90 (fatal) is generated.
The procedure is called after every NEW (inserted by PAP)

Description of the procedure NEWSTK.

Heading

```
PROCEDURE NEWSTK(QID:QPOINTER;PRI:INTEGER;HL:INTEGER;SL:INTEGER)
```

Task of the procedure

Generating a new process descriptor and new process stacks.
Saving the registers R4, R5, R6, the linenumber, and the pdpointer on the software stack.
Changing R5 and R6 to the new created stacks.
Initializing the field NR of the new process descriptor.

Parameter description

QID: pointer to the processor queue of the process for which a process descriptor is made
PRI: priority of that process
HL : size of the hardware stack to be created
SL : size of the software stack to be created

Called procedures and functions

CHREG
NEWCHK

Calling procedures and functions

none

Suggestions for extensions

none

Further remarks

The user process START is replaced by NEWSTK and a call of the process to be started by PAP.

Description of the procedure OVFLCHK.

Heading

PROCEDURE OVFLCHK

Task of the procedure

Testing if a process hardware or software stack overflow occurs.

Parameter description

none

Called procedures and functions

$WRERROR

Calling procedures and functions

INIT

Suggestions for extensions

none

Further remarks

If hardware stack overflow occurs an error 91 (fatal) is generated.
If software stack overflow occurs an error 92 (fatal) is generated.
The procedure uses the copy of the pdpointer in the local memory.
The call of OVFLCHK is inserted after each procedure or function declaration.

Description of the procedure PARAMERROR.

Heading

PROCEDURE PARAMERROR(NR:INTEGER;KIND:INTEGER;PARAM:INTEGER)

Task of the procedure

generate an error message with one extra parameter

Parameter description

NR : the number of the error to be generated

KIND : defines the seriousness of the error
(1 = fatal, 2 = warning, 4 = message)

PARAM: the parameter displayed in the error message (e.g. offending value)

Called procedures and functions

$WRERROR

Calling procedures and functions

STRINGTOINTEGER

Suggestions for extensions

none

Further remarks

The procedure can be used in RSX programs too.

Description of the procedure PHOENIX.

Heading

PROCEDURE PHOENIX(VAR PROCAQ:QUEUE;VAR PROCBQ:QUEUE;VAR READYQ:QUEUE)

Task of the procedure

- Placing the main program in the readyqueue.
  (The main program is supposed to do no I/O)
- Filling a data area with its APR's, registers 3, 4, 6, memory management register 0 and the processorqueues.
- Creating the trap routines for TRAP TO 0, 4, 10, 14, 20, 24, 250.
- Map the virtual addresses 0-17777B to the corresponding absolute addresses.
- Copying the setup routine for the other processors to a part of the common memory that is addressable with memory management diabled. This routine initializes the APR's and registers of a processor and gives the processor its own processor queue address, wakes the first process, and initializes the trap interrupt vectors with the addresses of the interrupt routines.

Parameter description

PROCAQ: the processor bound queue for one processor
PROCBQ: the processor bound queue for the other processor
READYQ: the queue which can be used by both processors.
(Remark: the procedure is written for two processors)

Called procedures and functions

WAKEUP
QENTER

Calling procedures and functions

none, the procedure is called in the main program (inserted by PAP)

Suggestions for extensions

none

Further remarks

The trap routines for traps to 0, 4, 10, 14, 20, 24, 250 place the address of the trap vector in R0 and execute a HALT instruction (in KERNEL mode). If the routines are proceeded (with the console ODT command P) a RTI is executed.
Remember that R0 is destroyed at that moment!
The start address of processor A is (at the moment) 20100, that of processor B 20200.

Description of the procedure QENTER.

Heading

PROCEDURE QENTER(VAR Q:QUEUE;VAR PDPOINTER:PDPT)

Task of the procedure

Placing a process in a queue with slotword, according to the priority of the process

Parameter description

Q : queue in which the process is going to be placed

PDPOINTER: pointer to the process descriptor of the process that must be queued

Called procedures and functions

BSARPS

SPSABW

QIN

Calling procedures and functions

SIGNAL

RETTOM

PHOENIX

Suggestions for extensions

none

Further remarks

QIN places a process in a queue without paying attention to the slotword. QENTER does pay attention to the slotword.

Description of the procedure QIN.

Heading

PROCEDURE QIN(VAR GET:PDPT;VAR PDPOINTER:PDPT)

Task of the procedure

Placing a process in a queue according to the priority of the process

Parameter description

GET : pointer to the first process descriptor in the queue

PDPOINTER: pointer to the process descriptor of the process that must be queued.

Called procedures and functions

none

Calling procedures and functions

QENTER

CHANGE

Suggestions for extensions

none

Further remarks

none

Description of the procedure QLEAVE.

Heading

PROCEDURE QLEAVE(VAR Q:QUEUE;VAR PDPOINTER:PDPT)

Task of the procedure

wake a process from a queue with slotword

Parameter description

Q : queue from which a process is going to be awoken

PDPOINTER: pointer to the process descriptor which is removed from the queue

Called procedures and functions

SPSABW

BSARPS

QOUT

Calling procedures and functions

WAKEUP

Suggestions for extensions

none

Further remarks

QOUT wakes a process without paying attention to the slotword,
QLEAVE does pay attention to the slotword.
If after execution PDPOINTER = nil the queue is empty.

Description of the procedure QOUT.

Heading

PROCEDURE QOUT(VAR GET:PDPT;VAR PDPOINTER:PDPT)

Task of the procedure

wake a process from a queue

Parameter description

GET : pointer to the first process descriptor in the queue

PDPOINTER: pointer to the process descriptor which is removed from the queue

Called procedures and functions

none

Calling procedures and functions

QLEAVE

SIGNAL

Suggestions for extensions

none

Further remarks

see under QLEAVE

Description of the function READREG.

Heading

FUNCTION READREG(PART:INTEGER;REG:ADDRESS):WORD

Task of the function

reading a virtual memory address
the contents of the address is the function result

Parameter description

PART : tells in which part of 4k words we want to read
(PART = 0: between 0 and 4k etc.)
REG : Gives an address inside these 4k words (13 bit)
READREG: contents of the address represented as an integer

Called procedures and functions

none

Calling procedures and functions

none

Suggestions for extensions

a better way for parameter transport

Further remarks

If the function result is between 0 and 77 777B, it is represented between 0 and 32767. Values between 100 000B and 177 777B are represented between -32768 and -1 resp.
For this reason the address is splitted in two parts.

Description of the procedure RECEIVE.

Heading

```
PROCEDURE RECEIVE(VAR PDPOINTER:PDPT;VAR MSG:UNIV ARRAY [INTEGER]
                                        OF CHAR;VAR MBUF:UNIV MESBUF)
```

Task of the procedure

receiving a message from a buffer.

Parameter description

PDPOINTER: pointer to the process descriptor of the calling process
MSG : contains after procedure execution the received message
MBUF : the buffer where the message is received from

Called procedures and functions

WAIT
SIGNAL
COPYFROM

Calling procedures and functions

none

Suggestions for extensions

none

Further ermarks

see under SEND

Description of the procedure RESPS.

Heading

PROCEDURE RESPS(PS:INTEGER)

Task of the procedure

restoring the processor status with the contents of variable PS

Parameter description

PS: contains the processor status, the calling process should get

Called procedures and functions

none

Calling procedures and functions

BSARPS

Suggestions for extensions

Perhaps the variable PS should be cleared after restoring the processor status

Further remarks

none

Description of the procedure RETTOM.

Heading

```
PROCEDURE RETTOM(VAR INQUEUE:QUEUE;VAR R5:INTEGER)
```

Task of the procedure

- place the calling process in INQUEUE
- return to the program part that the calling process started. This happens with help of the contents of R5

Parameter description

INQUEUE: the queue, in which the calling process is placed

R5 : contents of register 5 of the program part that started the calling process

Called procedures and functions

QENTER

Calling procedures and functions

INIT

Suggestions for extensions

none

Further remarks

none

Description of the procedure SAVPS.

Heading

PROCEDURE SAVPS(VAR PS:INTEGER)

Task of the procedure

saving the processor status in variable PS

Parameter description

PS: contains after procedure execution the processor status of the calling process

Called procedures and functions

none

Calling procedures and functions

SPSABW

Suggestions for extensions

none

Further remarks

none

Description of the procedure SEND.

Heading

```
PROCEDURE SEND(VAR PDPOINTER:PDPT;VAR MSG:UNIV ARRAY [INTEGER]
                                    OF BYTE;VAR MBUF:UNIV MESBUF)
```

Task of the procedure

placing a message in a buffer

Parameter description

PDPOINTER: pointer to the process descriptor of the calling process

MSG : the message to be transported

MBUF : the buffer where the message is send to

Called procedures and functions

WAIT

SIGNAL

COPYTO

Calling procedures and functions

$WRERROR

Suggestions for extensions

none

Further remarks

The procedure uses the reserved word UNIV.

Therefore we are not bounded by the array lengths declared in the heading.

Description of the procedure SIGNAL.

Heading

PROCEDURE SIGNAL(VAR SEM:SEMAF)

Task of the procedure

If there are processes sleeping in the semaphore queue, one of them is awoken and placed in its processor queue. Otherwise the semaphore counter is incremented.

Parameter description

SEM: the semaphore which must be signalled

Called procedures and functions

SPSABW

BSARPS

QOUT

QENTER

Calling procedures and functions

SEND

RECEIVE

ATTMAC (in the generated interrupt service routine)

Suggestions for extensions

none

Further remarks

none

Description of the procedure SIMPLEERROR.

Heading

PROCEDURE SIMPLEERROR(NR:INTEGER;KIND:INTEGER)

Task of the procedure

Generate an error message

Parameter description

NR : the number of the error to be generated

KIND: defines the seriousness of the error

(1 = fatal, 2 = warning, 4 = message)

Called procedures and functions

$WRERROR

Calling procedures and functions

STRINGTOINTEGER

INTEGERTOSTRING

Suggestions for extensions

none

Further remarks

The procedure can be used in RSX programs too.

Description of the procedure SPSABW.

Heading

PROCEDURE SPSABW(VAR SLOT:WORD;VAR PS:INTEGER)

Task of the procedure

saving the processor status in variable PS, and executing a busy wait on the variable slot

Parameter description

SLOT: word, which controls access to a queue

PS : contains the processor status of the calling process after execution of the procedure

Called procedures and functions

SAVPS

BWAIT

Calling procedures and functions

SIGNAL

QLEAVE

BYE

WAIT

QENTER

Suggestions for extensions

none

Further remarks

The processor status is on absolute address 777 776B, which corresponds with virtual address 177 776B.

SPSABW means Save Processor Status And Busy Wait.

Description of the procedure STRINGTOINTEGER.

Heading

PROCEDURE STRINGTOINTEGER(STRING S;VAR I:INTEGER)

Task of the procedure

Converting an array of CHAR (or part of it) to an integer

Parameter description

S: array or array part to be converted

I: conversion result

Called procedures and functions

PARAMERROR

SIMPLEERROR

Calling procedures and functions

none

Suggestions for extensions

none

Furhter remarks

Blanks in the string are ignored.

If an illegal character is detected (no blank, sign or digit) an error message 94 is generated. The ascii value of the character is given in this message. As conversion result a zero is given.

If overflow is detected MAXINT or MININT (depends on the sign already detected) is given as conversion result, and an error message 95 is generated.

Description of the procedure WAIT.

Heading

PROCEDURE WAIT(VAR PDPOINTER:PDPT;VAR SEM:SEMAF)

Task of the procedure

placing the calling process in a queue if the semaphore counter is zero, otherwise decrement the semaphore counter and continue

Parameter description

PDPOINTER: pointer to the process descriptor of the calling process

SEM : the semaphore on which must be waited

Called procedures and functions

SPSABW

BSARPS

CHANGE

Calling procedures and functions

SEND

RECEIVE

Suggestions for extensions

none

Further remarks

none

Description of the procedure WAKEUP.

Heading

PROCEDURE WAKEUP(VAR Q:QUEUE;VAR PDPOINTER:PDPT)

Task of the procedure

wake a process. If possible a process from Q is awoken, otherwise a process from the READYQ. This is tried until a process is awake.

Parameter description

Q : queue from which a process is going to be awoken, if not empty

PDPOINTER: pointer to the process descriptor of the process which is awoken

Called procedures and functions

QLEAVE

Calling procedures and functions

PHOENIX

CHANGE

Suggestions for extensions

none

Further remarks

none

Description of the procedure WRITEREG.

Heading

PROCEDURE WRITEREG(PART:INTEGER;REG:ADDRESS;CONT:WORD)

Task of the procedure

writing on a virtual memory address

Parameter description

PART: tells in which part of 4k words we want to write
(PART = 0: between 0 and 4k etc.)
REG : gives an address inside these 4k words (13 bit)
CONT: the word which must be written to the address

Called procedures and functions

none

Calling procedures and functions

none

Suggestions for extensions

A better way to transport the address

Further remarks

see under READREG

Description of the routine $WRERROR.

Heading

none

Task of the routine

sending an error message to ERRORBUF if needed (determined by the dynamic option switch word 4(R3))
If a fatal error occurs and a user doesn't want to continue after a fatal error a BYE is executed.

Parameter description

see listing

Called procedures and functions

SEND
BYE

Calling procedures and functions

NEWCHK
OVFLCHK
PARAMERROR
SIMPLEERROR
all Pascal runtime routines that generate error messages

Suggestions for extensions

none

Furhter remarks

Errorbuf address is in 2(R3)
The format of an error message can be found under the type descriptions
The routine is based on the pascal runtime routine $WRERROR

Description of the standard processes.

Some processes will nearly always be used. To make the use of the system more convenient for a user, a few of these processes are already implemented as standard processes. All of them have reentrant code, so they can be started multiple times. On the following pages these processes are described in the same way the procedures and functions are described, although somewhat more comprehensive. Because processes can't have calling procedures and functions that section is omitted.

Description of the standard process TTIN.

Heading

PROCESS TTIN(VAR CHARBUF:UNIV CHBUFTYPE)

Task of the process

The process reads characters from the console terminal (address 177 562B). The highest bit of each byte is ignored, so all characters are of the type ASCII. The characters read are placed in the buffer CHARBUF, which is a process parameter.
Errors of the interface, from which the characters are read, are ignored.

Parameter description

CHARBUF: A buffer where all characters received from the interface are send to. The buffer should have messages of one element of type ASCII or BYTE.

Called procedures and functions

ATTACH
WRITEREG
WAIT
SEND
READREG

Suggestions for extensions

none

Further remarks

The process assumes that the interface from which the characters are received, has its status register at address 177 560B, and its data register at address 177 562B. Also the interface must be vectored via address 60B.
Estimated hardware stacksize required is 15 words.
Estimated software stacksize required is 70 words.
The process is CPU bound.

Description of the standard process TTUIT.

Heading

PROCESS TTUIT(VAR CHARBUF:UNIV CHBUFTYPE)

Task of the process

The process writes characters to the console terminal (address 177 566B). All bits are transmitted. The characters which are written are received from the buffer CHARBUF, which is a process parameter. If a carriage return is received from the buffer a linefeed is automatically added.

Parameter description

CHARBUF: A buffer where all characters, which are written, are received from. The buffer should have messages of one element of type ASCII, CHAR or BYTE.

Called procedures and functions

ATTACH
WRITEREG
RECEIVE
WAIT

Suggestions for extensions

none

Further remarks

The process assumes that the interface, to which the characters are written, has its status register at address 177 564B and its data register at address 177 566B. Also the interface must be vectored via address 64B.
Estimated hardware stacksize is 15 words.
Estimated software stacksize is 70 words.
The process is CPU bound.

Description of the standard process ERRORLOG.

Heading

PROCESS ERRORLOG(VAR BUF:UNIV CHBUFTYPE)

Task of the process

The process receives error messages from the buffer ERRORBUF. These messages are split into characters which are send to the buffer BUF, which is a process parameter. The messages are preceded and followed by a carriage return.

Parameter description

BUF: A buffer where all characters are send to. The buffer should have messages of one element of type ASCII or BYTE.

Called procedures and functions

RECEIVE

SEND

Suggestions for extensions

none

Further remarks

It is recommanded to start the process with a high priority. Also the buffer which is given as parameter can be coupled to a TTUIT process directly.

Estimated hardware stacksize required is 15 words.

Estimated software stacksize required is 70 words.

The process is not CPU bound.

Appendix F: Description of the compiler modification.

In contrary with the design considerations mentioned in chapter 2.1., one compiler modification was made. This was necessary, because otherwise it was impossible to report the linenumber where an error occured (see chapter 4.2.).
The modification made stores the linenumber at an address in local memory (now 772B), instead of 2(R3). The modification only concerns the compiler procedure LINENODEF. In this procedure the lines

```
GEN2(MOV,AUTINC,PC,INDEX,GP);
GENCONST(LINENO);GENCONST(LINEADDR);
```

which generate the instruction MOV #lineno,2(R3) are changed to

```
GEN2(MOV,AUTINC,PC,AUTINCDEF,PC);
GENCONST(LINENO);GENCONST(772B);
```

which generates the instruction MOV #lineno, @#772

Appendix G: Software changes needed for hardware changes.

The software is rather flexible with regard to hardware changes. Some changes however need a small software modification. These are (for the most common cases) mentioned below.
The hardware can be expanded with another processor. The software system must then be extended with a processor queue for this new processor. This queue must be declared in the texts inserted by the preprocessor, and in the system software. Also the procedure INITIALIZE must be extended to initialize this new queue. Finally PHOENIX must be modified to give the new processor its own initialization routine. This also implies that the constant NROFPROCESSORS must be updated, and that the new queue must be a parameter of PHOENIX, so that the declaration and call of PHOENIX inserted by the preprocessor must be changed too.
It is also possible to change the amount of local memory given to each processor. In that case the constants BEGPAR and BEGADR from PHOENIX must be changed. Also in PHOENIX the APRs via which the local memory is mapped, must be initialized properly. The TKB option STACK should be given the size of the local memory in words, when taskbuilding a task for the multiprocessor system.

Appendix H: Some remarks on Pascal.

During the development of the system, some errors in the Ericsson Pascal compiler are detected. These are reported here. Some of these errors are already reported earlier (ref. 7), but will be included here too.

- Because character arrays are always packed (so contain 2 characters per word), one should be careful when comparing arrays of odd length. Comparison of the elements is done on word basis, so comparing identical arrays of odd length can yield the value FALSE, because the last (unseen) byte of both arrays differs.
- It is impossible to get a listfile when switches are placed after the object file.
- If a variable is declared in the procedure heading as
  S:ARRAY [INTEGER] OF CHAR
  one cannot use the statement WRITE(TTY,S) in that procedure.
- One cannot use the file TTY in separate compiled procedures, unless it is included in the procedure heading, and therefore must be included at every procedure call.
- It was impossible to create a working memory resident overlaid compiler. All the compilers that were build memory resident trapped when used.
- The H-switch doesn't operate as was expected. Occurences of the H-switch after the first BEGIN of the main program have no influence at all. When starting a program the dynamic option switch word (hidden variable 4(R3) is initialized according to the last occurrence of the H-switch before the first BEGIN of the main program, even if this last occurrence was in a procedure declaration. During program execution the dynamic option switch word is never changed.
- Message 41 and the corresponding IORESULT are not generated if number 32768. is entered. Entering too negative numbers results in reading number -77777B instead of MININT (-100000B). Numbers between 32769 and 65535 and between -32769 and -65535 are not changed into MAXINT resp. MININT, but in the number read before. Multiple attempts result in a system trap.
  All these errors are caused by errors in the routine RDI, which can

be found in the file P11RDI.MAC (in PACLIB.PAC). The routine RDI is changed, and all errors mentioned above are corrected.

- The compiler doesn't detect a missing semicolon between an assignment statement and an IF or WHILE statement (other statements not investigated).
- It is nearly impossible to read an array of characters from file TTY, else than character for character.
- Using -32768 in ones program gives the compiler error 203 (integer constant exceeds range).

Appendix I: Modifications made in the "practicumsysteem".

At the THE a course "Procescomputersystemen" is given (ref.20). Parallel to this course students can write software for a small process computer system (usually called "practicumsysteem) as exercise. During the last years this software was written in assembler for the multi-processor system. It is intended to use the Pascal system for that purpose. A try-out and demonstration version of the "practicumsysteem" is already written (ref. 7). This demonstration program is used as test program during the development of the Pascal system. This led to the detection of some minor errors in the demonstration program. Also some greater improvements were made. These are described below.

- All assembler procedures are replaced by calls of READREG or WRITEREG.
- Starting the processes is implemented.
- The types MESSAGE and MSGBUF are used.
- Some calls of SEND and RECEIVE are changed to transmit integers instead of character arrays.
- The process REKENPROCES is rewritten.
  This is done because the ADC gives 12 bit samples instead of 8 bit ones.
- The processes TEXT1TOCHAR and TEXT2TOCHAR, TTY1TOCHAR and TTY2TOCHAR, TTY1FROMCHAR and TTY2FROMCHAR are replaced by processes TEXTTOCHAR, TTYTOCHAR and TTYFROMCHAR. These processes are started twice (with shared code).
- The standard procedures STRINGTOINTEGER and INTEGERTOSTRING are used.